CSE 374
Programming Concepts & Tools

Hal Perkins

Winter 2012

Lecture 3 – I/O Redirection, Shell Scripts

Standard I/O streams and redirection

- Recall: every command has 3 standard streams: stdin (0, input), stdout (1, output), stderr (2, error messages)
- Default is keyboard (stdin), screen (stdout, stderr)
- Can redirect to a file with <, >

  echo hello > there

  cat < there; cat <there > here
- Can “pipe” output (stdout) of one command to input (stdin) of another with |

  man bash | less
- Done entirely in the shell – programs are oblivious; they just use streams 0,1,2

File redirection in (more) detail

- Somewhat cryptic; some common usages:
  - redirect input: cmd < file
  - redirect output, overwriting file: cmd > file
  - redirect output, appending to file: cmd >> file
  - redirect error output: cmd 2> file
  - redirect output and error output to file: cmd &> file
  - ...

  See bash manual sec. 3.6 for other variations
- Useful special file: /dev/null
  - Immediate eof if read; data discarded if written

Pipes

cmd1 | cmd2

- Change the stdout of cmd1 and the stdin of cmd2 to be the same, new stream!
- Very powerful idea:
  - In the shell, larger command out of smaller commands
  - To the user, combine small programs to get more usefulness
    - Each program can do one thing and do it well!
- Examples:
  - ps aux | less
  - djpeg me.jpg | pnmscale -xysize 100 150 | cjpeg > thumb.jpg

Combining commands

- Combining simpler commands to form more complicated ones is very programming-like. In addition to pipes, we have:

  cmd1 ; cmd2 (sequence)

  cmd1 || cmd2 (or, using int result – the "exit status" – run cmd2 if cmd1 "fails")

  - Example: do_something || echo "Didn't work!"

  cmd1 && cmd2 (and, like or; run cmd2 only if cmd1 "succeeds" – i.e., "returns" 0)

  - Example: check_if_ok && launch_missles

  cmd1 ‘cmd2‘ (use output of cmd2 as input to cmd1). (Note cmd2 surrounded by backquotes, not regular quotes)

  - Useless example: cd ‘pwd‘.
  - Non-useless example: mkdir ‘whoami‘A‘whoami‘.

(Non)-alphabet soup

- List of characters with special (before program/built-in runs) meaning is growing: ‘ ! % & * ~ ? [ ] " ’ \ > < | $ (and we’re not done).
- If you ever want these characters or (space) in something like an argument, you need some form of escaping; each of " ’ \ have slightly different meaning.
- First approximation:
  - "stuff" treats stuff as a single argument but allows some substitutions for $variables.

    example: cat "to-do list" # filename with spaces(!)
  - ’stuff’ suppresses basically all substitutions and treats stuff literally.

Shell Expansion and Programs

- Important but sometimes overlooked point: shell metacharacter expansion, I/O redirection, etc. are done by the shell before a program is launched
  - The program usually never knows if stdin/stdout are connected to the keyboard/screen or files
  - Program doesn’t see original command line – just expanded version as a list of arguments
  - Expansion is uniform for all programs since it’s done in one place – the shell

Shell as a programming language

- The shell is an interpreter for a strange programming language (of the same name). So far:
  - “Shell programs” are program names and arguments
  - The interpreter runs the program (passing it the arguments), prints any output, and prints another prompt. The program can affect the file-system, send mail, open windows, etc.
  - “Builtins” such as cd, exit give directions to the interpreter.
  - The shell interprets lots of funny characters differently, rather than pass them as options to programs.
- It’s actually even more complicated:
  - (two kinds of) variables.
  - some programming constructs (conditionals, loops, etc.)

Toward Scripts…

- A running shell has a state, i.e., a current
  - working directory
  - user
  - collection of aliases
  - History
  - Streams (files, etc.)
  - ...
- In fact, next time we will learn how to extend this state with new shell variables.
- We learned that source can execute a file's contents, which can affect the shell's state.

Running a script

- What if we want to run a bunch of commands without changing our shell's state?
- Answer: start a new shell (sharing our stdin, stdout, stderr), run the commands in it, and exit.
- Better answer: Automate this process.
  - A shell script as a program (user doesn't even know it's a script).
  - Now we'll want the shell to end up being a programming language
  - But it will be a bad one except for simple things

Writing a script

- Make the first line exactly: #!/bin/bash
- Give yourself “execute” permission on the file
- Run it
  - Probably need to precede filename with ./ if current directory isn’t normally searched for commands (i.e., ‘.’ is not normally included in $PATH – and it shouldn’t be for security reasons)
- Note: The shell consults the first line of the file:
  - If a shell-program is there, launch it and run the script (similar trick works for perl, python, etc.)
  - Else if it’s a “real executable” run it (more later)
- Example: listhome

More expressions

- bash expressions can be:
  - math or string tests (e.g., -lt)
  - logic (&&, ||, !) (if you use double-brackets)
  - file tests (very common; see Pocket Guide)
  - math (if you use double-parens)
- Gotcha: parens and brackets must have spaces before and after them!
- Example: dcdls (double cd and ls) can check that arguments are directories
- Exercise: script that replaces older file with newer one
- Exercise: make up your own

Accessing arguments

- The script accesses the arguments with $i to get the ith one (name of program is $0).
  - Example: make thumbnail1
- Also very useful for homework: shift (manual Section 4.1)
  - Example: countdown
- We would like optional arguments and/or usage messages. Need:
  - way to find out the number of arguments
  - a conditional
  - some stuff we already have
  - Example: make thumbnail2

Review

- The shell runs programs and builtins, interpreting special characters for filenames, history, I/O redirection.
- Some builtins like if support rudimentary programming.
- A script is a program to its user, but is written using shell commands.
- So the shell language is okay for interaction and “quick-and-dirty” programs, making it a strange beast.
- For both, shell *variables* are extremely useful.

Preview: Variables

```
i=17 # no spaces
set
echo $i
set | grep i
echo $i
unset i
echo $i
f1=$1
```

- (The last is very useful in scripts before shifting)
- Enough for next homework (arithmetic, conditionals, shift, variables, redirection, ...)
- Gotcha: using undefined variables (e.g., because of typo) doesn't fail (just the empty string).